CSE 2017 Data Structures and Lab

Lecture #9: Expression Tree

Eun Man Choi

A Binary Expression Tree is . . .

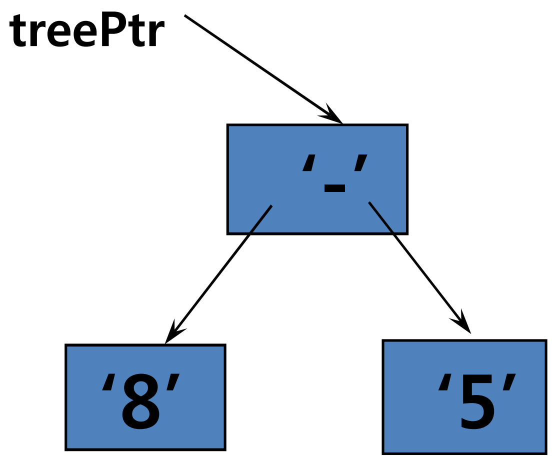
A special kind of binary tree in which:

1. Each leaf node contains a single operand,

2. Each nonleaf node contains a single binary operator, and

3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

A Two-Level Binary Expression



INORDER TRAVERSAL: 8 - 5 has value 3

PREORDER TRAVERSAL: - 8 5

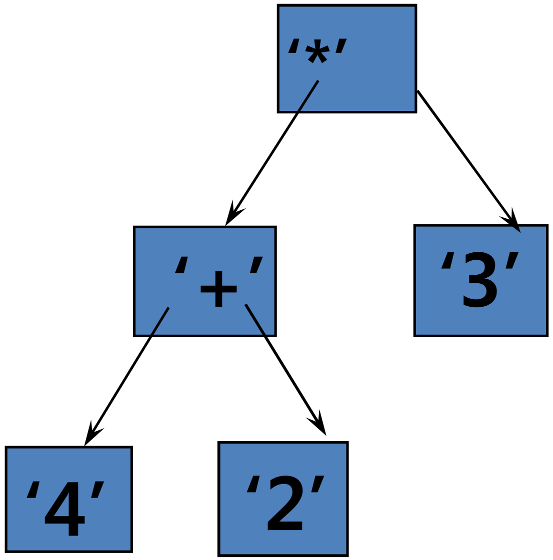
POSTORDER TRAVERSAL: 8 5 -

Levels Indicate Precedence

When a binary expression tree is used to represent an expression, the levels of the nodes in the tree indicate their relative precedence of evaluation.

Operations at higher levels of the tree are evaluated later than those below them. The operation at the root is always the last operation performed.

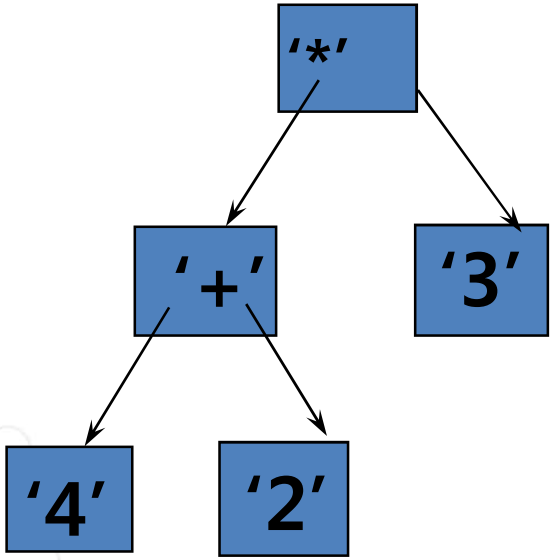
A Binary Expression Tree



What value does it have?

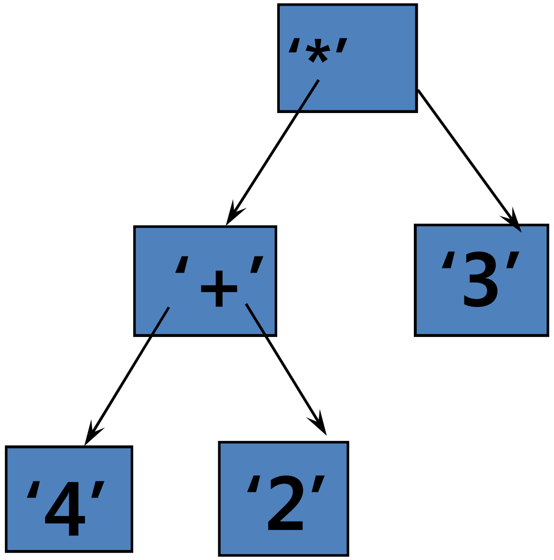
(4 + 2) * 3 = 18

A Binary Expression Tree



What infix, prefix, postfix expressions does it represent?

A Binary Expression Tree

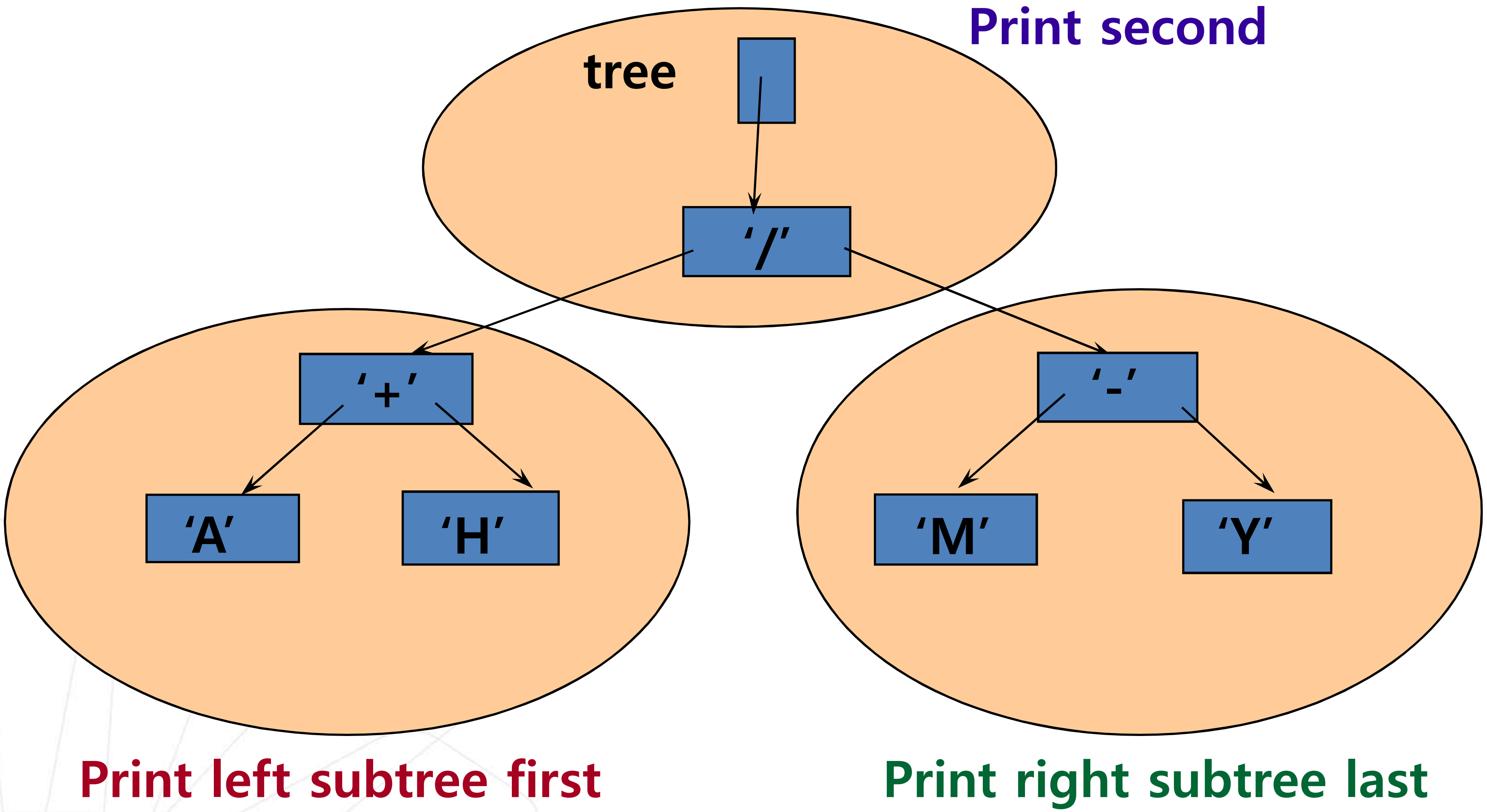


Infix: ((4 + 2) * 3)

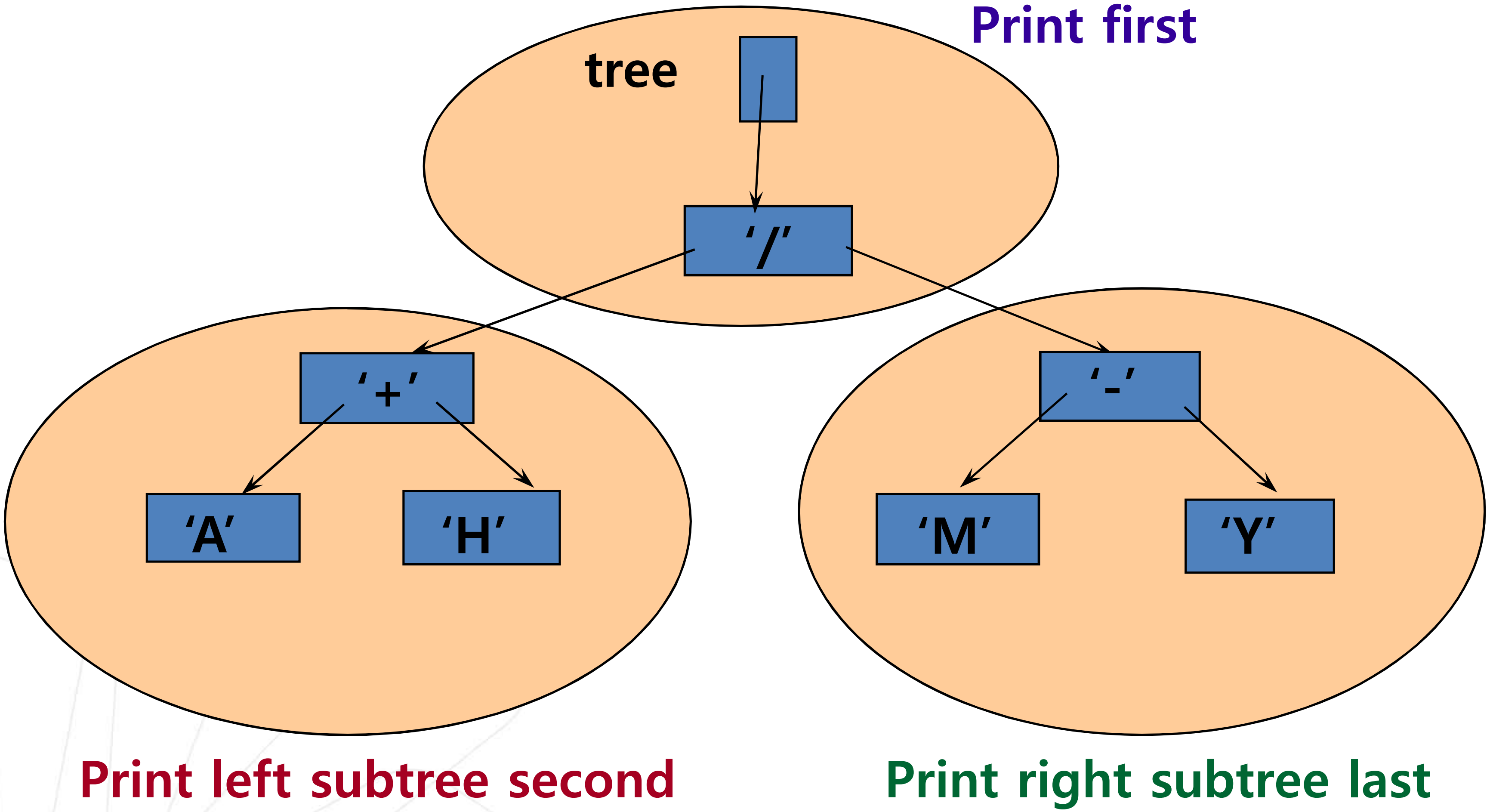
Prefix: * + 4 2 3

Postfix: 4 2 + 3 * *has operators in order used*

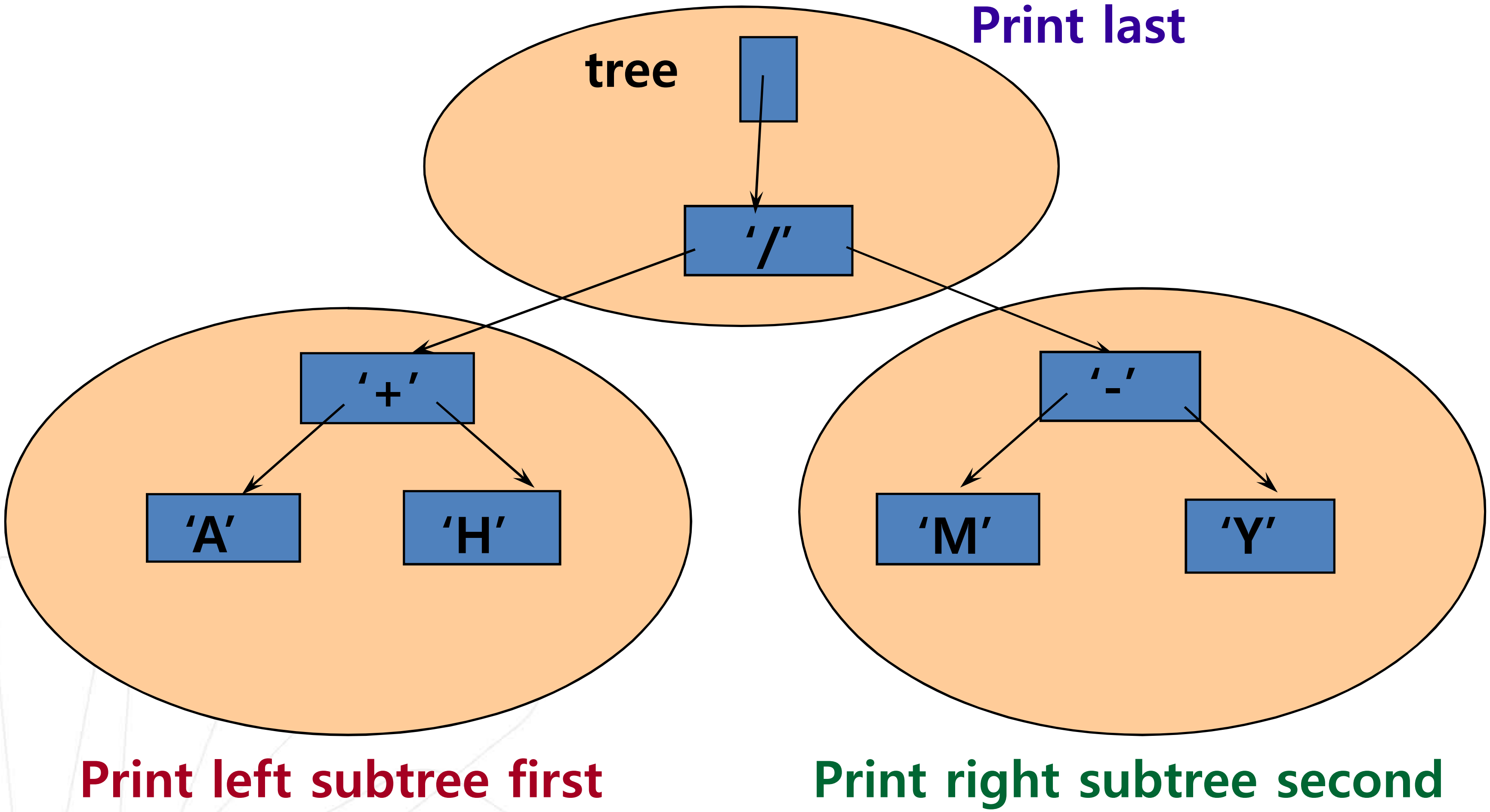
Inorder Traversal: (A + H) / (M - Y)



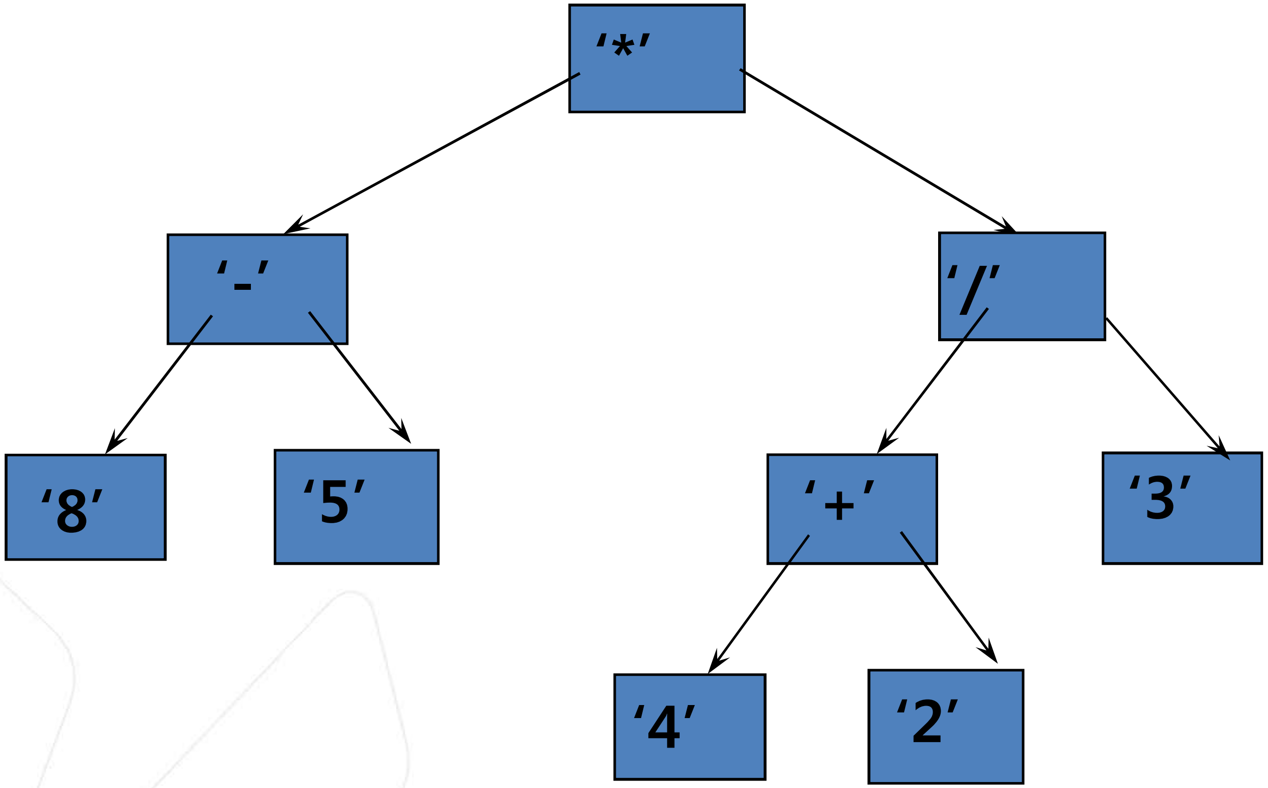
Preorder Traversal: / + A H - M Y



Postorder Traversal: A H + M Y - /

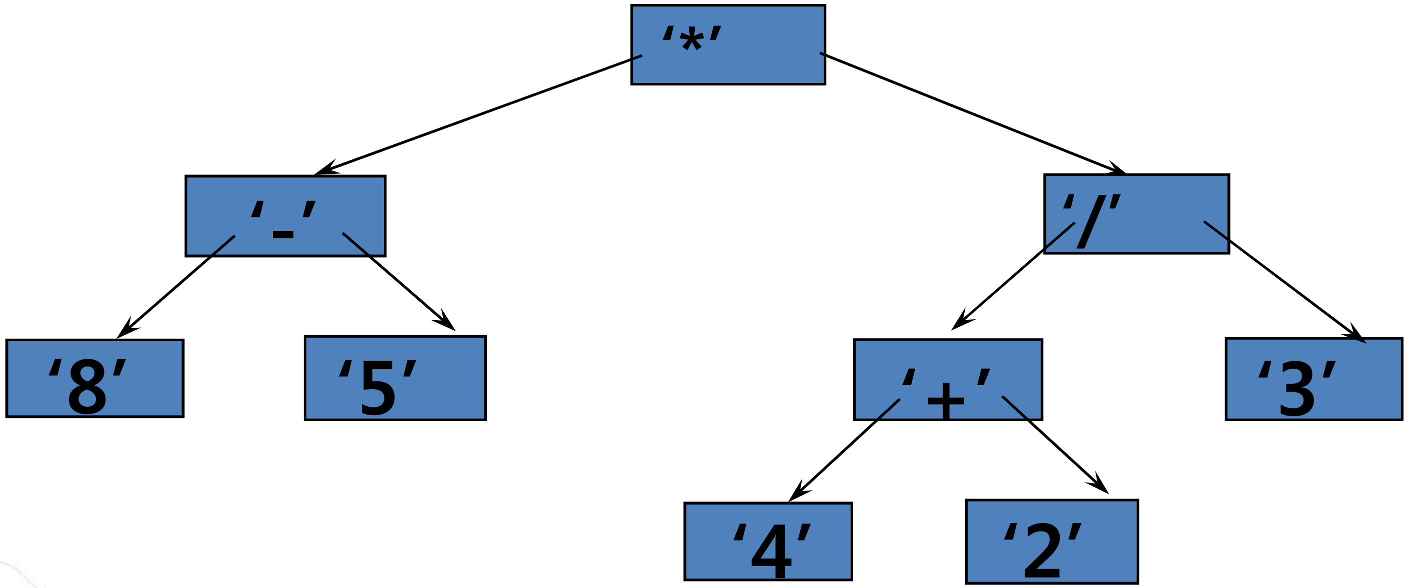


Evaluate this binary expression tree



What infix, prefix, postfix expressions does it represent?

A binary expression tree



Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / * *has operators in order used*

동국대학교 dongguk university

ExprTreeNode (Lab 11)

```
class   ExprTreeNode  {
   private:
      ExprTreeNode (char elem,
      ExprTreeNode *leftPtr, ExprTreeNode *rightPtr); // Constructor

      char           element;   // Expression tree element
      ExprTreeNode  *left,      // Pointer to the left child
                    *right;     // Pointer to the right child
      friend class Exprtree;
};
```

NULL	*	6000
. left	. element	. right

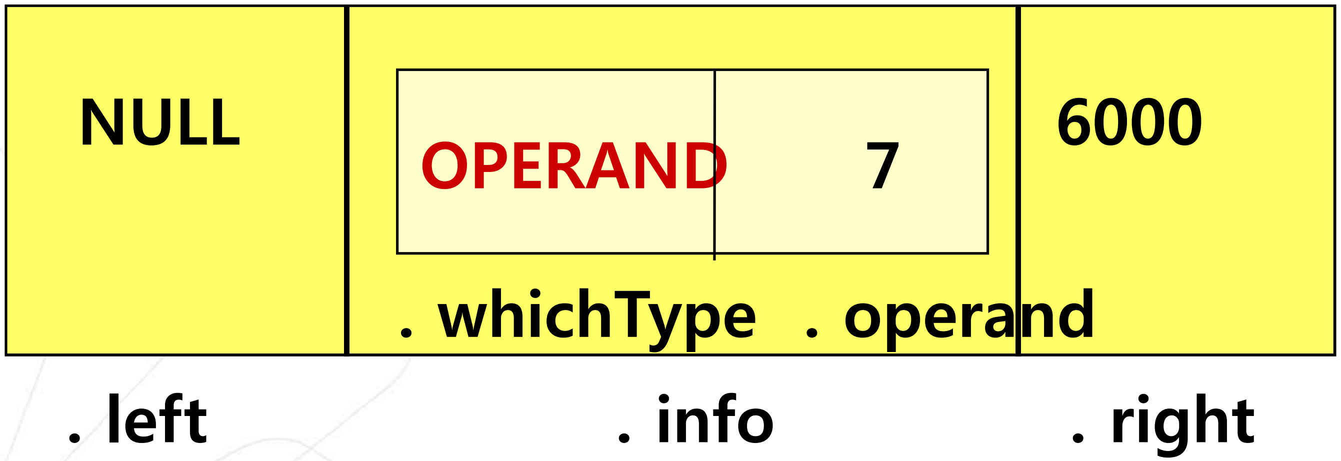
InfoNode has 2 forms

```
enum  OpType  { OPERATOR,  OPERAND } ;
struct   InfoNode   {
     OpType          whichType;
     union                              // ANONYMOUS union
     {
        char     operation ;
        int         operand ;
     }

};
```

OPERATOR	'+'		OPERAND	7
. whichType	. operation		. whichType	. operand

Each node contains two pointers

```
struct   TreeNode  {

    InfoNode      info ;           // Data member
    TreeNode*    left ;            // Pointer to left child
    TreeNode*    right ;           // Pointer to right child
};
```



Function Eval()

- **Definition**: Evaluates the expression represented by the binary tree.
- **Size**: The number of nodes in the tree.
- **Base Case**: If the content of the node is an operand,
  Func_value = the value of the operand.
- **General Case**: If the content of the node is an operator BinOperator,

Func_value = Eval(left subtree)
BinOperator Eval(right subtree)

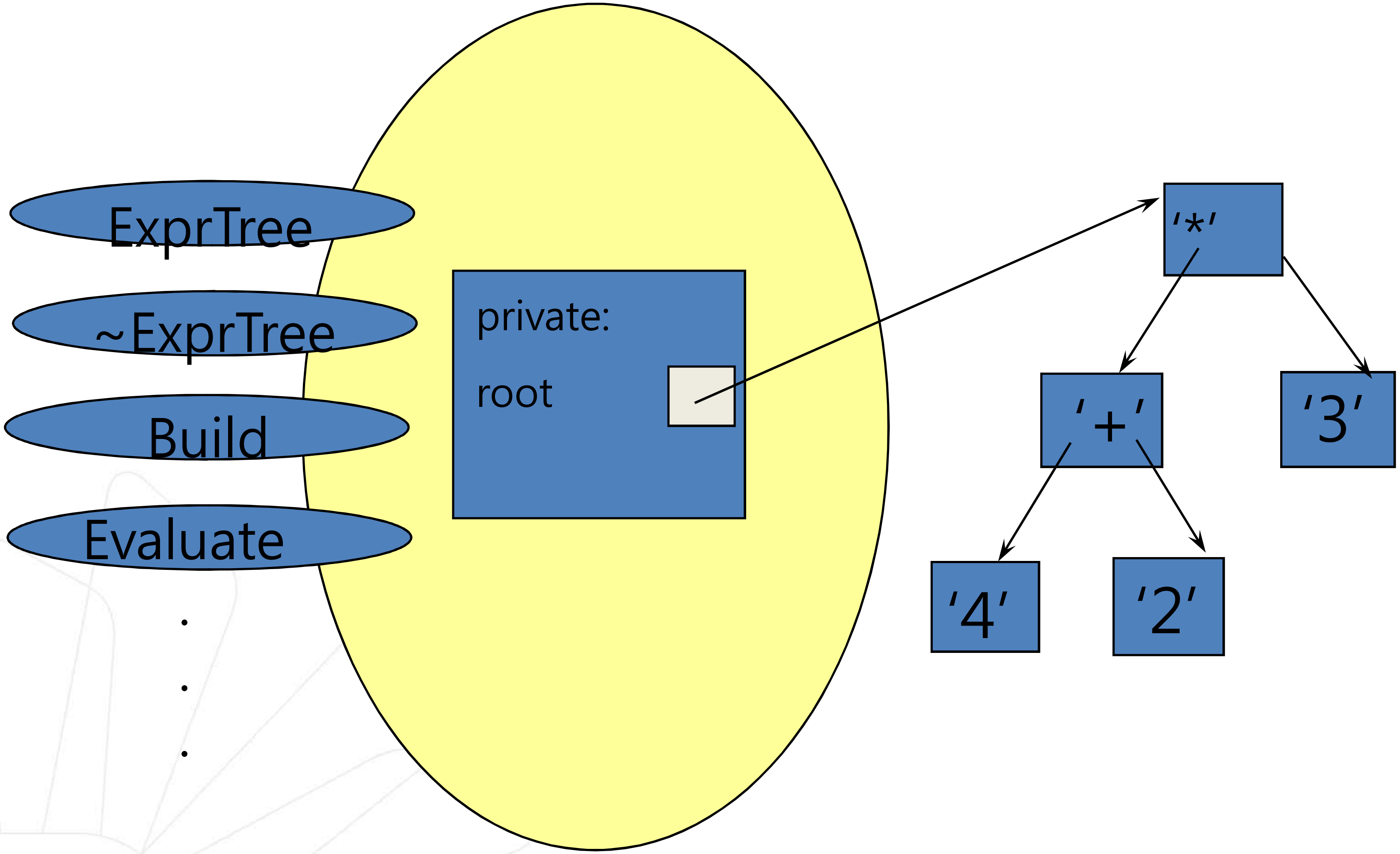
Eval(TreeNode * tree)

Algorithm:

```
IF Info(tree) is an operand
  Return Info(tree)
ELSE
  SWITCH(Info(tree))
    case + :Return Eval(Left(tree)) + Eval(Right(tree))
    case - : Return Eval(Left(tree)) - Eval(Right(tree))
    case * : Return Eval(Left(tree)) * Eval(Right(tree))
    case / : Return Eval(Left(tree)) / Eval(Right(tree))
```

```
int   Eval ( TreeNode*  ptr )

// Pre:     ptr is a pointer to a binary expression tree.
// Post:   Function value = the value of the expression
represented
//        by the binary tree pointed to by ptr.

{   switch  ( ptr->info.whichType ) {
    case OPERAND :  return  ptr->info.operand ;
    case OPERATOR :
        switch ( tree->info.operation ) {
          case '+': return(Eval(ptr->left) + Eval(ptr->right));
          case '-': return(Eval(ptr->left) - Eval(ptr->right));
          case '*': return(Eval(ptr->left) * Eval(ptr->right));
          case '/': return(Eval(ptr->left) / Eval(ptr->right));
        }
    }
}
```

class ExprTree



Build()

```
void ExprTree::build (){
  char *prefix = new char[20];
  cin >> prefix;
  BuildSub(root, prefix);
}

void ExprTree::BuildSub(ExprTreeNode *&ptr, char *&szExpr){
  ExprTreeNode *t;
  while(*szExpr){
        t = new ExprTreeNode;
        t->element = *szExpr;
        ptr = t;
```

Build() cont.

```
        if(is_operator(*szExpr)){
                BuildSub(ptr->right, ++szExpr);
                BuildSub(ptr->left, ++szExpr);
                return;
        }
        else {
                return;
        }
    }
  }
```

Expression()

```
void ExprTree::expression () const{
  ExpressionSub(root);
}

void ExprTree::ExpressionSub(ExprTreeNode *p) const{
  if(p != 0){
        cout << '(';
        ExpressionSub(p->left);
        cout << p->element;
        ExpressionSub(p->right);
        cout << ')';
  }
}
```